Aalto University
School of Science
Master's Programme in Engineering Physics

Ville Haikola

Optimization of a Search Function in a Large Software Product

Master's Thesis
Espoo, May 25, 2016

Supervisor: Professor Juha Karhunen
Advisor: Iikka Olli M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in Engineering Physics

ABSTRACT OF
MASTER'S THESIS

Author: Ville Haikola

Title:
Optimization of a Search Function in a Large Software Product

Date: May 25, 2016 **Pages:** 60

Professorship: Information and Computer Science **Code:** IL3010

Supervisor: Professor Juha Karhunen

Advisor: Iikka Olli M.Sc. (Tech.)

In this work we studied the Building information modelling software Tekla Structures. Our goal was to improve the performance of a specific functionality in the case study software: the filtering of model objects. Filters are a set of rules customizable by the user. By applying filters user is able to filter model objects by their properties, for example by name, length or construction date. Filters are used in many key processes in Tekla Structures such as drawing creation, selection and how objects are visualized.

We started by investigating the current implementation of filtering and where the performance could be improved. In our methods we are partly restricted by the large size of the software product, which makes high level changes difficult to implement. After investigation we were able to find more efficient algorithms and data structures to significantly improve the equation processing related to filtering. We found significant differences in the calculation times of properties, which lead us to investigate ways to optimize the evaluation order of the rules in the filter. Finding the optimal evaluation plan is in its general form a NP-hard combinatorial optimization problem.

We then reimplemented the equation processing related to filtering. This included implementation of several algorithms to optimize the evaluation order of rules in the filter. These algorithms included a full exhaustive search of all evaluation plans and computationally less expensive methods such as an algorithm based on boolean differential algebra. The performance improvement was then calculated with user created structural models and in more controlled simulations. The reimplementation of the equation processing alone improved the performance of the filtering by more than 50% in our tests without the optimization of the evaluation order. The evaluation order optimization also gave significant improvement to the performance, which is more apparent in filters with a large number of rules. All the implemented algorithms were able to improve the performance and the computationally less expensive methods were almost as effective as the full exhaustive search of all evaluation plans.

Keywords: BIM, Filtering, JOP, Performance analysis, Boolean difference

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Teknillisen fysiikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Ville Haikola

Työn nimi:
Hakutoiminnon optimointi suuressa ohjelmistotuotteessa

Päiväys:	25. toukokuuta 2016	**Sivumäärä:**	60
Professuuri:	Tietojenkäsittelytiede	**Koodi:**	IL3010
Valvoja:	Professori Juha Karhunen		
Ohjaaja:	Diplomi-insinööri Iikka Olli		

Tässä työssä tutkittiin rakennusten mallinnukseen käytettävää Tekla Structures -ohjelmistoa. Tavoitteena oli parantaa mallikappaleiden suodatukseen liittyvää toimintoa. Suodatin Tekla Structures -ohjelmistossa koostuu säännöistä, joiden perusteella voi rajoittaa mallikappaleiden joukkoa. Rajoitus tapahtuu kappaleiden ominaisuuksien kuten nimen, pituuden tai rakennuspäivämäärän perusteella. Suodattimia käytetään useassa eri tärkeässä prosessissa, kuten esimerkiksi piirustusten luonnissa ja kappaleiden visualisoinnissa kolmiulotteisissa malleissa.

Aluksi tutustuttiin suodattimen nykyiseen toteutukseen. Tehokkuusanalyysin perusteella löydettiin algoritmeja ja tietorakenteita, joiden perusteella tehokkuutta voitiin parantaa. Ohjelmiston laajamittaisessa muokkaamisessa on haasteena sen suuri koko, monia miljoonia koodirivejä.

Mallikappaleiden ominaisuuksien laskenta-ajoissa havaittiin isoa vaihtelua. Tämän vuoksi tutkimme voisiko suodattimen sääntöjen laskujärjestystä optimoimalla vaikuttaa sen suoritusaikaan. Optimaalisen laskujärjestyksen löytäminen on NP-täydellinen optimointiongelma. Tutkimme vastaaviin ongelmiin, kuten tietokantojen taulujen yhdistämisjärjestykseen käytettyjä algoritmeja.

Tutkimusten perusteella päätimme toteuttaa suodattimeen liittyvän yhtälönratkaisijan uudelleen. Toteutimme algoritmeja, joilla pystyimme vaikuttamaan sääntöjen suoritusjärjestykseen. Toteutettuja algoritmeja olivat yksinkertainen säännön hintaan perustuva uudelleenjärjestys, boolean differentiaalialgebraan perustuva menetelmä sekä kaikki mahdolliset suoritusjärjestykset läpikäyvä haku. Uuden yhtälönratkaisijan tehokkuutta mitattiin testeissä, joita tehtiin aidoilla asiakasmalleilla sekä simuloidulla datalla. Toteutettujen muutosten myötä suodattimen suoritusaika pieneni yli 50% testeissä jo ilman suoritusjärjestyksen optimointia. Suoritusjärjestyksen optimointi paransi suoritusaikaa merkittävästi kaikilla toteutetuilla algoritmeilla. Laskennallisesti vähemmän vaativat algoritmit olivat lähes yhtä tehokkaita kuin kaikki mahdolliset suoritusjärjestykset läpikäyvä haku.

Asiasanat:	BIM, Suodatus, JOP, Suoritusanalyysi, Boolean difference
Kieli:	Englanti

Acknowledgements

I wish to thank my company Trimble for giving me an interesting thesis subject and for the chance to work with the related implementation and writing tasks on my working hours. This made the completion of the thesis possible in a reasonable time. I thank my supervisor Juha Karhunen for his fast review of my thesis and my advisor Iikka Olli for his guidance throughout the project. I also like to thank my team members for their support and enduring my absence from the team's duties for the duration of this project. Finally, I would like to thank my friends and family for their support.

Espoo, May 25, 2016

Ville Haikola

Abbreviations and Acronyms

BB	Branch and Bound
BD	Boolean Difference
BDD	Binary Decision Diagram
BIM	Building Information Modelling
JOP	Join Ordering Problem
TS	Tekla Structures
UDA	User defined attribute.

Contents

Chapter 1

Introduction

Advances in information technology has transformed the tools of structural design from pen and paper to advanced Building Information Modelling(BIM) software of today. Users can create detailed 3D models that can be used throughout the whole building life cycle from design and detailing to maintenance. Tekla Structures, the case study software in this thesis, is a BIM software that has a wide international user base. The users are able to build models with high level of detail from the largest beams and columns down to individual bolts and welds. [1]

The goal of this work is to improve the performance of a specific functionality in Tekla Structures: the filtering of model objects. Filters consists of set of conditions fully customizable by the end user. Objects can be filtered by their properties such as material, length or construction date. Filters are frequently created and applied to model objects in many separate processes in Tekla Structures. This includes how objects in the 3D modelling view are visualized, creation of a 2D drawing and what objects the user is able select from model. Significant improvement in filtering speed will thus improve the software performance in many key areas. Current implementation of filtering slows down the software performance in large models where the filters can be in some cases applied to millions of objects at once. Ways to improve the performance should be investigated, as the size and level of detail of customer models is ever increasing.

Problem statement

This work is a case study where we seek to significantly improve the performance of one specific feature in Tekla Structures, the filtering of model objects. We study the current implementation with an eye on what kind of

algorithms and data structures would be needed to improve the performance. In our work we are partly restricted by the large size of the software product. Not everything will be possible to implement in the scope of this work if it requires high level modifications to the software.

After examination it becomes clear that the current implementation of equation library in Tekla Structures is not optimal for solving the boolean expression in the filtering process. We will rewrite the equation processing using efficient data structures. Optimizing the evaluation order of the rules in the filter expression is important to study, as we found significant differences in the calculation times of different properties in our analysis. This problem is related to the join ordering problem that is encountered in the database query optimization. In its general form, the join ordering problem is a generalization of the classical travelling salesman problem and thus NP-complete. We will review different optimization algorithms commonly used for the problem and pick methods to be implemented to our software product. The implemented methods will need to be thoroughly tested and analysed before they can be added as a part of the international software product.

Structure of the Thesis

The thesis begins with an introduction to the case study software. The emphasis is on filtering and finding the key areas where its performance depends on. In chapter 3 where we review methods that could be useful for the performance optimization based on our findings. In particular we review methods commonly used for optimizing boolean expression and the join ordering problem. In chapter 4 we describe the implementation of our chosen methods to Tekla Structures in detail. In chapter 5 we describe the testing of our implementation and analyse the results. Finally in the discussion and conclusions we summarize the results, how we met our objectives and discuss possible ways to go forward.

Chapter 2

Background

Company

Trimble is a company based in the United States that works in a number of different industries including geospatial solutions, agriculture and construction. What is now the Trimble Solutions Finland was formerly known as the Tekla corporation, a Finnish software company established in 1966 and acquired by Trimble in 2012. The products developed in the Trimble Solutions Finland include software solutions for energy distribution, infrastructure management and construction. Most of the development is centred in the Espoo headquarters where about 400 employees are working. The most important product is the Tekla Structrues, which we will introduce in the next section. [1, 2]

Tekla Structures

Tekla structures(TS) is a large software product designed for Building Information Modelling(BIM). Customers can use the product throughout the building process for conceptual design, detailing, fabrication and construction management. Software is used for creating 3D structural models as well 2D drawings. A picture of the 3D Modelling view can be seen in figure 2.1 [1]

The models created with Tekla Structures can be constructed with a high level of detail. The models can consist of a large selection of building elements such as beams, columns, plates and reinforcing bars. The parts can be connected in many ways, for example welded or bolted together. The part geometry can be modified by for example cutting or chamfering a part. All types of materials can be used, the most common materials being steel

and concrete. There exists also tools for construction management, such as monitoring the construction dates for individual objects and assemblies. The size of the model extend to the size of stadiums and skyscrapers consisting of millions of objects.

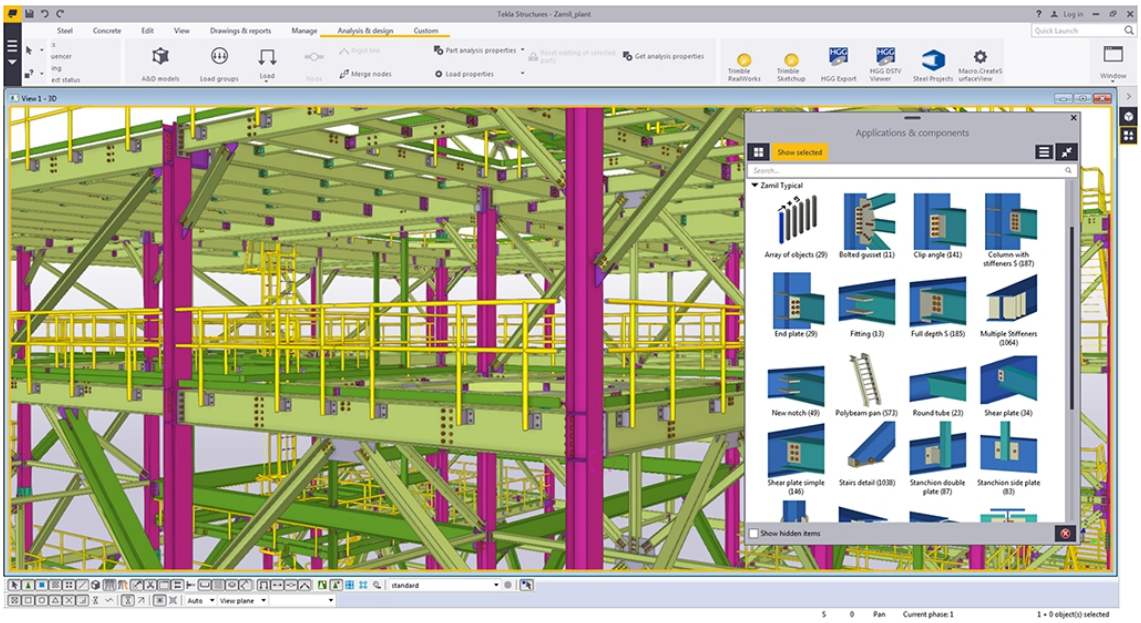


Figure 2.1: A 3D modelling view in the Tekla Structures software. [1]

The development of the software has been ongoing for more than 25 years. Currently the source code consists of more than 6 million lines of code written in the C, C++ and C# programming languages. A hundred developers are working on the software to create new features, fix defects and improve the software architecture and performance.

Filtering

Filters consists of set of conditions fully customizable by the end user. Picture 2.2 shows how a selection filter can be constructed in Tekla Structures. For each filter row, customer first selects a category, which can be for example `Part`, `Assembly`, `Reinforcement Bar` or many others. This will determine the type of model objects that the filter row can be applied to. Also for some categories such as `Task` or `Assembly`, the filtering will be applied to related objects, such as all the tasks that are linked to the object. For each category

there exists a set of properties user can choose from. For certain categories such as `Part` this list can be more than a hundred items long and include user defined properties. User can then select from a list of conditions including equals, greater than, begins with and many others depending on the type of the property which can be string, integer, floating point number or a date. Finally user can set the expected value. This is the value to which the value of the property in a model object is compared.

The return value of the whole filter for one model object is always either true or false. Filter rows can be joined together by `and` or `or` conditions, where `and` has precedence to `or` in the evaluation order. Also user can freely add parenthesis to the equation. In a typical filter, only few rows are required, but sometimes there might be up to ten and in rare cases even more rows.

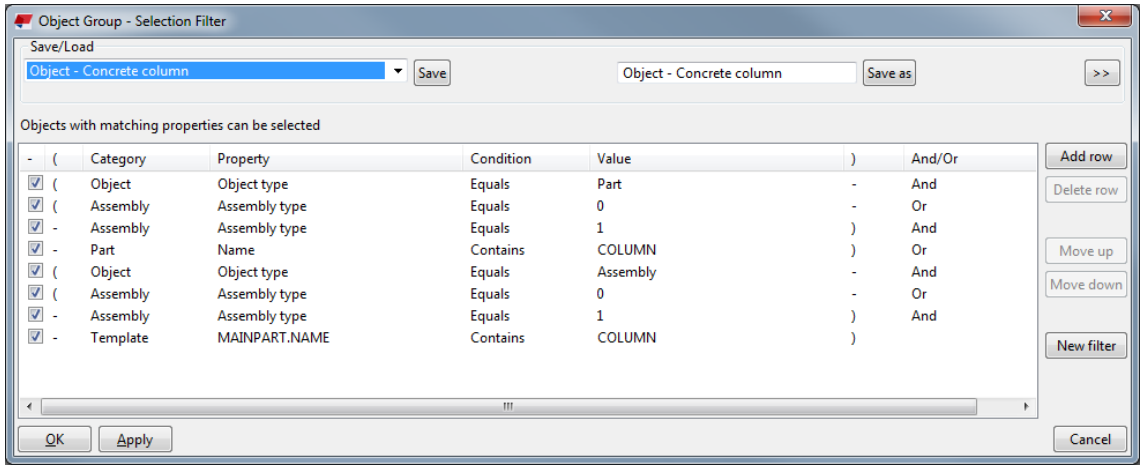


Figure 2.2: The user interface, through which selection filters can be created.

The Current Filter Implementation

Major operations related to filters are the creation of the filter in the user interface, initialization and applying the filter to a single model object. After user defines a filter in the UI, the input is saved in the disk and it can be used to the end of the project.

Initialization is done every time a process is executed that requires the filter. In the initialization, the user input is processed by the filter class. In the current implementation, this includes creation of equation strings as required by the external equation library tool. For each row in the filter, an equation string is created and the whole filter is described as an equation

string containing series of if-then-else commands, which the equation library is able to parse.

Once a filter is initialized it can be applied from one to millions of model objects depending depending on the process. This is typically the most performance critical part. For each object that needs filtered, the function `IsObjectFiltered` is called. The function gets the object id as a parameter and returns `true` if the model object passes the filter. For example when user selects a set objects in the model, a selection filter is initialized. System fetches the ids of all the selected model objects. For each selected object id the function `IsObjectFiltered` is called and only the objects that pass the filter criteria are set as selected.

`IsObjectFiltered` function fetches the related object ids for each filter row if this is required by the category, for example all the tasks related to the object. The equation library receives the equation strings and the ids and starts the evaluation of the filter. The filter rows are executed in straightforward order from top to bottom. All of the filter rows are not executed if the return value is already determined by the preceding rows. The equation library is able to access the Tekla Structures property interface to receive the value of the property for given model object id.

Property Management in Tekla Structures

Modelling the structure of buildings and work flow of a construction project require a rich and customizable attribute system. Model objects in Tekla structures can have hundreds of different properties. The performance of the property interface has a key role in the filter performance.

The core of the Tekla Structures property management system is the `DbVirtual` virtual database management system, which has been used since the beginning of development and is built in house. The data is stored in memory buffers, which increases the performance significantly when compared to more traditional database systems.

In the source code of TS, model objects including for example parts, assemblies and bolts belong to a hierarchy of objects derived from the base class `CommonObject`. When an object is created and selected from the database many operations are performed, most noticeably data is selected from multiple database tables.

Properties of objects are typically fetched via the property interface in the TS model module. For this to happen, a `CommonObject` has to be created and selected from the database. Interface class then starts the process to fetch the property. Sometimes fetching a property might require property specific operations which could take more time than the object creation.

Each model object can also have their own user defined attributes(UDA), which are stored in separate database tables depending on the type, which can be a string, an integer or a floating point value.

In cases such as filtering where specific data is needed from a potentially huge number of objects, the current property interface is not optimal. A more optimal solution would be to use the design pattern called lazy loading, so that only the data required by the operation is fetched, and parts of the object would remain in uninitialized state. This would, however, require a significant amount of time consuming refactoring work. However, for some frequently used properties such as `name`, `type` and `material` direct database access has been allowed for the filtering system. These properties can be calculated a lot faster than with the properties interface as the `CommonObject` creation can be bypassed.

Performance of Fetching Properties

To better understand the performance issues related to filters we need to study the time it takes to fetch properties for the filtering system. We are expecting to see major variance in the speed depending on the property. A large model representing a bridge was studied as an example. The model contained more 21646 objects classified as parts.

In the test program, the part ids were fetched from the database. All the properties for the filter categories `part`, `object` and `assembly` were chosen for the analysis, more than 300 properties in total. All of the properties were then fetched for all the parts in the model. Timers were placed in the tester code to measure the performance of each property. The results sorted by execution time where then plotted in figure 2.3. In the figure, one dot represents a single property in the system, and the y-axis represents the average time it takes to calculate the property for a model part.

As you can see in the plot, there is a handful of properties that are very fast and a few that are significantly slower than others. A large majority of properties are in the middle and take about the same time $15\mu s$ to calculate. This is due to the overhead created by the property interface and the `CommonObject` creation. Many of these properties are user defined attributes.

The fast properties are dominated in number by the slower ones, but most of fast ones are used very frequently in filters. For example the property `name` takes only $0.5\mu s$ to fetch directly from the database and it is the most common property used in filters. On the slow side there is also some commonly used properties such as `profile`, which takes about $40\mu s$ to calculate in this model. For some assembly and task related properties also the related objects are fetched and checked by the filtering system, which is not

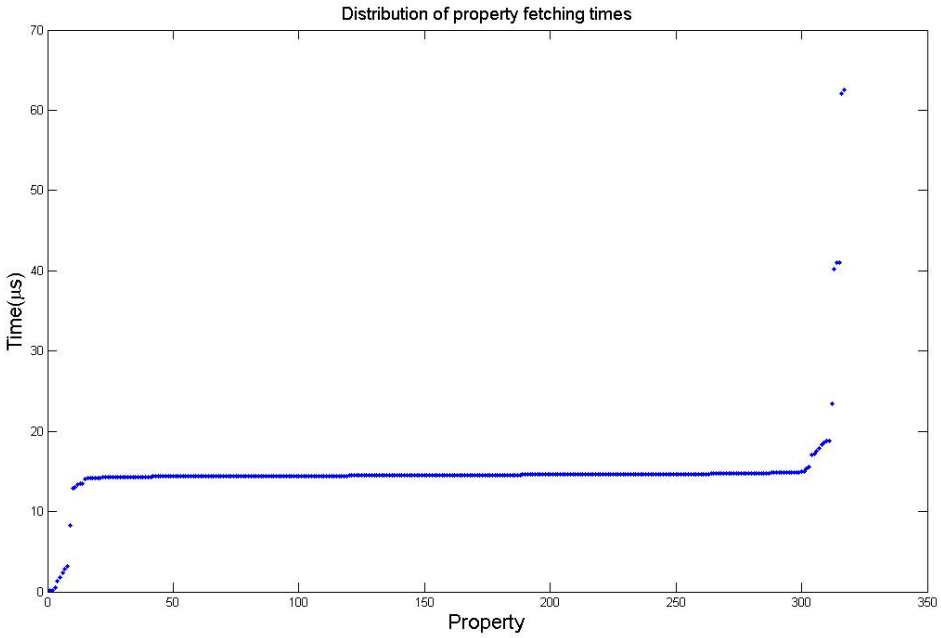


Figure 2.3: The distribution of the property calculation costs for an example model. One dot represents a single property in the system, and the y-axis represents the average time it takes to calculate the property for a single model part

shown in this analysis.

Based on this analysis there is at least two orders of magnitude difference between the slowest and fastest properties frequently used in filters. This leads us to believe that optimizing the evaluation order of filtering might be very beneficial for the filtering performance. If a filter contains both properties `name` and `profile`, the `name` should obviously be calculated first.

How Filters Are Used by Customers

Farmi is an in house built automatic testing system for Tekla Structures. Large number of test cases using different structural models are automatically run to ensure high quality of the product. Much of the testing is done using real customer models and some models are generated by testers for testing specific key operations in the TS.

Searching through the Farmi model database we can find 2135 filters, which can give us a good indication of how the filters are used. These filters

have 6851 rows in total, which gives an average of 3.2 rows for each filter. in table 2.1 we can the row count distribution of filters. Most of the filters, 61%, are either one or two rows long. Approximately 96% are under ten rows, but a few longer ones exist. The largest filter found contained 26 rows. The Longest filters typically have some degree of repetition, so that the same properties are used in many different rows.

Table 2.1: The rule count distribution of the filters in Tekla Structures. Most of the filters have few rules, but even filters larger than 15 exist. Largest filter found in this analysis had rule count 26.

Number of rows	Number of filters	Ratio of all filters	Cumulative ratio
1	832	0.3897	0.3897
2	465	0.2178	0.6075
3	259	0.1213	0.7288
4	100	0.0468	0.7756
5	75	0.0351	0.8108
6	71	0.0333	0.8440
7	79	0.0370	0.8810
8	137	0.0642	0.9452
9	26	0.0122	0.9574
10	21	0.0098	0.9672
11	9	0.0042	0.9714
12	17	0.0080	0.9794
13	10	0.0047	0.9841
14	7	0.0028	0.9869
15	8	0.0037	0.9906

The Use of properties in filters was studied as well. In table 2.2 we can see the most commonly used properties in filters and the time it takes to calculate these properties. We can roughly categorize the properties in three groups

- 53% Fast database properties $0.2\mu s - 3\mu s$
- 5% Slow properties $> 40\mu s$
- 42% Middle properties, that all take about the same time $14.6\mu s$.

Table 2.2: The most commonly used properties in filters and the time it takes to calculate them. There is large variation in the calculation times.

Property name	Ratio	Calculation time(μs)
User defined attributes(many)	0.35	14.6
Name	0.23	0.51
Object Type	0.13	0.18
Class	0.088	3.14
Material	0.040	2.40
Serie	0.034	13.0
Phase	0.031	2.82
Profile	0.026	40.2
Id	0.013	0.18
Numbering Serie	0.011	13.5
Start number	0.0096	12.9
GUID	0.0077	14.0
Numbering Position	0.0063	41
Assembly type	0.0048	23.4

Approximately 93% of operators used in the filters are either equals or does not equal. `and` is the most common logical conjunction at 85% leaving 15% for `or`.

Chapter 3

Methods

Boolean expression representation

As we studied the current filtering implementation we found that the performance of the equation processing could be significantly improved. We will consider an efficient data structure to represent the boolean expression in the filter. There exists many different ways to represent a boolean equation which are suited for different purposes. These include for example the propositional directed acyclic graph and the negative normal form. The Binary decision diagram(BDD) is a clear compressed data structure that is a strong candidate because it is easy to store and traverse in a computer program. [3, 4]

A figure representing a BDD of a boolean expression $x_1 \vee x_2 x_3$ can be seen in figure 3.1. In BDD the boolean equation is visualized as a rooted directed acyclic graph where terminal nodes are either 0-terminal or 1-terminal nodes. Other nodes are decision nodes, which are labelled by a boolean variable x_i. Nodes will have two children, a high child and a low child. Low child corresponds to the case where boolean variable evaluates to true and high child where it is false. The return value of the function with the current value assignment can be calculated by traversing the tree from root to a terminal node.

BDD is considered to be ordered, if the variables appear in the same order. If all the isomorphic subgraphs have been removed and all the nodes whose two children are isomorphic are eliminated, the BDD is called reduced.

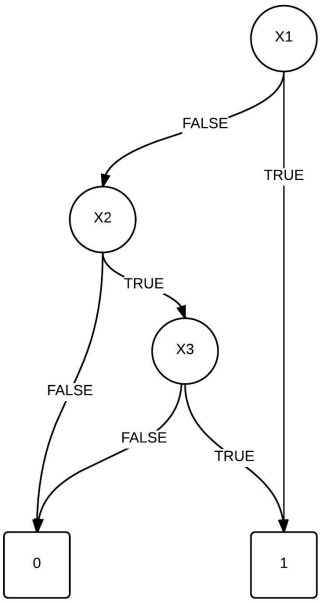


Figure 3.1: A visualization of a binary decision diagram of boolean expression $f(\mathbf{x}) = x_1 \vee x_2 x_3$. Each boolean variable is represented by a decision node with two children for the cases where the variable evaluates to true or false. The terminal nodes determine the end result of the function.

Join Ordering Problem

A problem similar to the optimal evaluation of the boolean expression in the TS filtering is often encountered in database systems. When user fetches data from a database this is done by writing a query in a query language such as `sql`. In a database query the user can determine a boolean expression as a `where` clause which restricts the desired result from all the records in the database. The data can be collected from various sources and accessed in different ways. Query optimizer turns the query to a cost effective evaluation plan. Processing times of the query may vary by many orders of magnitude depending on the evaluation plan. [5, 6]

The join ordering problem(JOP) is a difficult combinatorial optimization problem where many different algorithms have been used. The search space or solution space in the JOP is considered to be the set of evaluation plans that correspond to the same query. A solution can be described by a processing tree that evaluates the join expression. Sometimes restrictions can

be applied to the tree structure, for example considering only left deep trees, where $n!$ different solutions exist, where n is the number of joins. There exists a cost function, that maps a solution to a cost. The goal is to find the solution with the minimum possible cost. In its general form, the join ordering problem is a generalization of the classical travelling salesman problem and thus NP-complete.

The database systems typically use information of the database structure and statistics of the data stored in database to determine the cost of a solution. Information such as the size of the table accessed can be used. Cardinality describes the uniqueness of values in a database column, for example the column describing the sex of a customer would have lower cardinality than the name column. A histogram of values in a column can also be used in the optimization. This requires a process where statistics are recalculated in a regular manner either from the all the records of the table or a smaller sample. We are not considering cost functions in database systems in more detail as they are quite technical and not especially related to our case study.[8, 9]

Algorithms Used for the Join Ordering Problem

In general, all algorithms applicable to the NP-hard optimization problems will also be applicable for JOP. We will review some of the common algorithms used for the join ordering problem. [5, 6]

For smaller queries with ten or less joins, it typically still possible to do a complete traversal of the search space to find the global optimum. This is typically enough for most traditional database systems, but it is not feasible for some contemporary databases such as object oriented databases where tens or even hundreds of joins can be used. The approaches to simplify the problem can be classified to sacrificing generality and sacrificing exactness. The former method will add restrictions to the solution space such as consider only left deep trees and the latter method will occasionally accept suboptimal solutions for the problem. The algorithms for JOP can be roughly classified to four groups: deterministic, randomized, genetic and hybrid algorithms.

The deterministic algorithms perform a deterministic step by step search of the solutions space. The dynamic programming methods that perform a full traversal of the search space while possibly pruning unlikely solutions belong to this category. This method is used in almost all relational database systems where the number of join relations is usually quite small. One somewhat new deterministic method is based on relational difference calculus[8]. The goal of this method is to find the most significant relation based on the boolean difference calculus. We will expand on this method in section 3.3.

There is a large number of other deterministic algorithms, such as the A* search algorithm, which is efficient at pruning infeasible solutions.

Randomized algorithms are typically used for larger queries in less traditional databases where the number of joins can be large. A set of moves is defined, which constitute an edge between different solutions in the search space. A good move when considering only left deep trees can be for example swapping the join order of two relations. Method called `3Cycle` performs a cyclic rotation of three different relations.

Algorithms will then perform a random walk in the solution space starting from a randomly selected point. In the hill climbing algorithm, the best of all available moves is chosen. Iterative improvement algorithm selects a random neighbour and performs the move, if it improves the cost. It is usually less likely to get trapped in local minima and avoids the calculation of all possible moves, which can be costly because the number of neighbours can be very high. If no move that improves the cost is available, a local minimum is reached and the execution stops. The process is then repeated until suitable number of random starting points are considered or time limit is exceeded. The lowest local minimum is then chosen.

Simulated annealing algorithm is an improvement on the iterative improvement method, that is inspired by the natural annealing process of crystals. A concept of temperature is introduced, which describes the likelihood that a move leading to worse cost is accepted while cost improving moves are always accepted. The temperature is lowered as the minimization proceeds. When correctly executed, this algorithm is less likely to get trapped in local minima.

Hybrid algorithms try to combine the benefits of both randomized and deterministic methods. Fore example a method called toured simulated annealing performs the simulated annealing starting from deterministically chosen start points.

Genetic algorithms are quite similar to the randomized algorithms. Their unique characteristic is that they are inspired by the biological evolution. A random starting population generates offspring through crossover and mutation. The fittest solutions according to a cost function survive to the next generation. The process is repeated until some number of iterations or until the population gets homogeneous enough.

Boolean Difference Calculus

Boolean difference calculus[7, 8] can be used in a deterministic method to optimize boolean expressions. The main idea of the method is to find the

variable x_i with a best achievement / cost ratio. This will then be the first tested condition and after that other variables are considered. Boolean difference is defined as

$$\Delta_{x_i} f(\mathbf{x}) = f(x_0, x_1, \ldots, x_i = 0, \ldots, x_n) \oplus f(x_0, x_1, \ldots, x_i = 1, \ldots, x_n). \quad (3.1)$$

The probability that the boolean difference $p(\Delta_{x_i} f)$ is true is a measure of the impact of the variable x_i on the outcome of the whole boolean expression. If the variable x_i is is irrelevant to value of expression $f(\mathbf{x})$ then $\mathrm{p}(\Delta_{x_i} f)$ is zero and the value is one if the variable alone determines the result.

We will show how to calculate the boolean difference by considering a simple example function

$$f(\mathbf{x}) = x_0 \vee x_1 x_2. \quad (3.2)$$

To determine the boolean difference we have truth table of the function as an input to our algorithm in table 3.1. In this simple example we can clearly see from the function and the truth table, that the variable x_0 has the largest impact on the function result. The boolean difference of the variable x_0 as a function of x_1 and x_2 can be seen in table 3.2.

Table 3.1: The truth table corresponding to equation 3.2.

x_0	x_1	x_2	$f(\mathbf{x})$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Table 3.2: The boolean difference of variable x_0 for the truth table in 3.1

x_1	x_2	$\Delta_{x_0} f(\mathbf{x})$
0	0	1
0	1	1
1	0	1
1	1	0

If the probability of all variables is 0.5, then the probability of each row in table 3.2 is thus 0.5 * 0.5 = 0.25. The probability of the boolean difference $p(\Delta_{x_0} f(\mathbf{x}))$ in this case is thus 0.25 + 0.25 + 0.25 = 0.75. Using the same procedure, we can construct the boolean difference for variables x_1 and x_2, $p(\Delta_{x_1} f(\mathbf{x})) = p(\Delta_{x_2} f(\mathbf{x})) = 0.25$.

The Algorithm Based On Boolean Difference Calculus

An optimization method for boolean expressions with expensive tests based on the boolean difference calculus was proposed by Kemper[8] in 1992. The optimization method goes as follows. We define a cost function c, which gives the cost of testing each variable x_i. We also define a factor

$$s_i = \frac{p(\Delta_{x_i} f)}{c(x_i)}. \tag{3.3}$$

The larger the value of s_i, the better the achievement / cost ratio of testing variable x_i is. We can then create a decision tree describing the correct evaluation plan with following algorithm

1. if variable count $n = 0$, there is nothing to evaluate. If $n = 1$, test x_0.
2. For each x_i, calculate the factor s_i as in equation 3.3.
3. Choose the variable x_i with the highest factor s_i to be tested next.
4. Apply the algorithm recursively to the $x_i = 0$ and $x_i = 1$ branches with $n - 1$ free variables.

This method is much less computationally complex than a full search of the the solution space, where all variable candidates should be considered instead of just the one with the best factor s_i. The complexity is in how to calculate the factor s_i.

Branch and Bound Algorithm

Branch and Bound(BB) algorithm[10, 11] is one of the most commonly used algorithms for NP hard combinatorial optimization problems. It has also been applied to the optimization of boolean expressions with expensive tests[8].

BB is a deterministic method that traverses through the search space in a recursive manner. The method differs from the exhaustive brute force search

by introducing a function for calculating the lower bound of any candidate solution from a partial solution. At first a solution is found for the problem by using any heuristic. This will then be saved as the current best solution and the upper bound of all candidate solutions. A queue is initialized with a partial solutions where none of the variables are assigned. A following algorithm is then followed.

1. Pop a node from the queue
2. If the node represents a candidate solution and has lower cost than the current best solution, then save it as the best solution.
3. Else branch on the node to produce more nodes by recursively splitting the search space.
4. Calculate the lower bound for each node. If the node lower bound is lower thant the current best solution, insert the node to the queue. Go back to 1.

A difficult part is often to calculate a good estimation for the lower bound.

Chapter 4

Implementation

In this work we created a new filtering class for Tekla Structures to replace the old implementation and the external equation library. As a new feature we will implement different algorithms to optimize the evaluation order of the filter rules. The flow of the algorithms is as in figure 4.1.

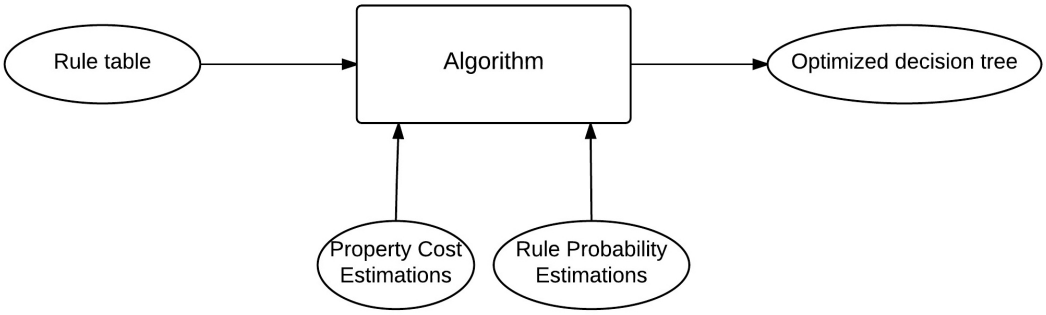


Figure 4.1: A Visualization of the evaluation order optimization algorithms.

The input for the algorithm is a table of rules that comes from the user input. The fields of a rule can be seen in table 4.1. The order of rules in the rule table defines the order in which the filter is evaluated in straightforward evaluation. Our implementation will transform this input to a decision tree that describes the evaluation order of the boolean expression. Algorithms can use the estimated property calculation costs and rule pass ratios generate the decision tree with the smallest average evaluation cost.

The output of the algorithm is a decision tree that consists of decision nodes. The contents of a decision node can be seen in table 4.2. This tree will then be traversed in the function `IsObjectFiltered` that gets the model

Table 4.1: The contents of a rule structure in Tekla Structures.

Field Name	Type	
Category Name	String	
Property Name	String	
Operator	String	for example equals, contains, greater than
Expected value	String	
Parenthesis open	Integer	an integer representing the level of parenthesis
Parenthesis closed	Integer	
And / Or	String	

Table 4.2: The contents of a Decision node structure in Tekla Structures.

Field Name	Type	
Type	enum	`DecisionNode`, `TrueNode` or `FalseNode`
Rule	Rule	
Left child	Decision node	
Right child	Decision node	

object id as parameter. The program will traverse the decision tree starting from the root.The evaluation goes as follows

1. Set the root of the decision tree as the current node.
2. If the node type is `TrueNode` return `true`. If it is `FalseNode`, return `false`
3. If the type of the model object does not fit with the rule's category, set the left child as the current node and go to step 2
4. Fetch the property in the rule for the model object if it is not already calculated previously.
5. Evaluate the expression by comparing the model object's property value to the expected value defined in the rule. If the expression returns `true` set the right child as the current node. Else, set the left child. Go to step 2.

The most complex part is the the initialization of the filter where the decision tree creation happens. Four algoritms were implemented, which we will go through in this chapter.

Calculation of costs and statistics

The creation of the optimal decision tree requires accurate statistics of the cost of the property calculation and probabilistic information of the pass ratio of the rules in filters. In our tests the costs were estimated for each property in the filter rule set before the filter creation process. This was done by fetching all the part ids in the model and calculating the property for all of them while calculating the passed time by timer in the code. A smaller sample of the model objects would likely have been enough for this analysis. An optimal sampling method might be one where the statistics are gathered from the same set of objects that the filter is applied to. Cost statistics should not be calculated on the fly every time a filter needs to be created as this eliminates all the potential gain from the better evaluation order. This however should be enough for our tests to see how much can be gained from accurate statistics and a more sensible way to implement the statistics calculations can be implemented later. A hard coded set of property costs could also be used, as calculation time doesn't seem to be dependent on the model for many properties, but this needs more investigation.

The histogram method was chosen for the probability estimations. The probability of the rule to be true was calculated from all parts of the model. In theory, this could be done daily as a background operation for commonly used filters either by calculating the properties for all objects or a smaller sample. Having accurate probability information for the model is a lot more difficult as they depend very strongly on the model and no hard coded values can be used. There also exists correlation between different properties. Variables can not thus be expected to be statistically independent

$$p(x_i = 1, x_j = 1)! = p(x_i = 1) * p(x_j = 1). \tag{4.1}$$

Calculation of correlations however would be quite a lot more time consuming especially for larger filters, where they should be calculated for each pair of variables. However, understanding the effect of probabilities have for the evaluation cost is still very valuable for the analysis.

Straightforward Evaluation

As a first step we will consider the creation the decision tree using the straightforward left-to-right evaluation. This is in principal how the current equation library implementation works, but reimplementing this using the decision tree model will likely give a performance improvement over the old implementation.

In Tekla structures filters the `and` operator has precedence over `or` operator and filters can contain many levels of paranhtesis. The basic formula for creating the decision tree from a list of rules is visualized in figure 4.2 and goes as follows.

1. Initialize a decision node for all the rules in the filter and the terminal `false` and `true` nodes.
2. Start From the first rule and the corresponding decision node.
3. Hook the node's true child to the next rule's node if operator is `and`. If there is no next node or the operator is `or` the node will be linked to terminal true node.
4. Hook the node's false child to next node following an `or` operator. If there is no `or` operator in the following nodes, the node will be linked to terminal false node.
5. Go to next Rule and go back to step 3. If there is no rules left, the algorithm is finished.

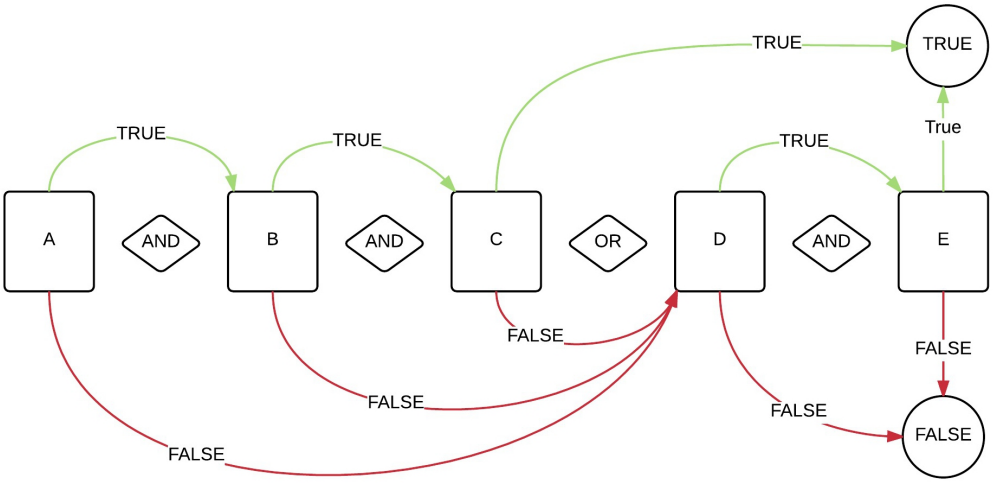


Figure 4.2: Visualization of how a decision tree can be generated using the straightforward evaluation order.

Parenthesis can be taken care of in a recursive manner by replacing them with a virtual node in the upper function and calculating a decision tree in

a similar fashion for the rule set inside parenthesis. The virtual node in the upper function is then replaced by the subtree. Any connections going to the virtual node will be hooked to the root of the subtree. Any connections inside the subtree that are linked to the terminal `true` and `false` nodes are then replaced by the virtual node's `right` and `left` children.

Simple Rearrangement Method

This method doesn't follow any general optimization algorithm for NP-hard problems, but it is designed specifically for this case. Knowledge of the cost or probabilistic information of rules can be used to reorder the rule list before the straightforward decision tree is generated. The Rule list is divided into sublists by the `or` operators. The rules inside the sublists, connected by the `and` operators, are then sorted by cost / pass ratio.

The evaluation order of these sublists can then be sorted by their own cost / pass ratio. The sorting factor for each sublist can be evaluated as in algorithm 3.

Algorithm 1 Get the sort factor of a sublist.

```
1: function GetSublistSortfactor(Sublist)
2:     Cost = 0
3:     Probability = 1;
4:     for Rule in Sublist do
5:         Cost = Cost + Probability * Rule.cost
6:         Probability = Probability * Rule.probability
       return Cost / Probability
```

Once again, the parenthesis can be handled by recursive methods. The cost of sorting arrays the size of even the largest rule tables is minimal. The benefit of this model is its simplicity and very fast execution. It can at least be used as comparison point for more advanced algorithms.

Boolean Difference Method

The Boolean Difference method was chosen as a candidate algorithm, because it seemed to be efficient and perhaps more reliable compared to the randomized algorithms. It is also straightforward to implement. The pseudo code of the implementation can be seen in algorithm 2. The decision tree creation was implemented in a breadth-first-search like manner using a queue.

The most significant rule will be chosen with algorithm 3. New node is created for the most significant rule and two new queue entries will be created for the true and false branches. Keeping track of already processed rules was done by introducing two arrays `LockedTrue` and `LockedFalse`. These were implemented as integers, where the bit corresponding to rule's index was set as 1, if the rule was processed. This means search and access operations are very fast and are not dependent on the rule table size as they were implemented using the bitwise and, or and negation operators. However, the number of rules has to be smaller than the size of integer in bits, either 32 or 64 bits.

The complex part is the calculation of the boolean difference itself. At first a truth table based method for calculating the boolean difference was implemented as proposed by Kemper[8]. However, the number of rows in the truth table is 2^N where N is the number of variables in the boolean function. Truth table based cost minimization methods are thus not efficient when the number of variables is large. A truth table for 30 rules already takes one gigabit of memory even if data is stored in the optimal way and traversing through an array likes this takes considerable time with today's computer power. The performance started to be unbearably slow as the number of rules in filter exceeded 15.

A following modification was made to improve the performance in boolean difference calculation. A supporting binary decision tree is first created by using the straightforward order. The value $p(\Delta_{x_i} f(\mathbf{x}))$ was then calculated for each variable as the difference between the probability of the boolean equation to be true in cases where x_i is true and where x_i is false. This was calculated by using the pre calculated decision tree as in the algorithm 4. This means that instead of generating and traversing the whole truth table with 2^N rows, we only need to traverse the decision tree generated from the straightforward order two times, once by locking $x_i = 0$ and once for $x_i = 1$. By definition this tree only contains N decision nodes and additional terminal nodes. This optimization can be justified, because the filters don't contain exclusive or conditions, and thus if the result of boolean equation is true when $x_i = 0$, it is also always true when $x_i = 1$, so the value for boolean difference will be the identical when calculates with this method.

Algorithm 2 Creates the decision tree using the boolean difference method. Requires a decision tree created using the straightforward or other method as input.

```
1: function CreateDecisionTreeBD(Rules, StraightforwardRoot)
2:     Root = null
3:     Q = new Queue
4:     FirstItem = new QueueItem
5:     FirstItem.ParentType = NoParent;
6:     FirstItem.ParentNode = null;
7:     FirstItem.LockedTrue = FirstItem.LockedFalse = ∅
8:     Q.push(FirstItem)
9:     while Q is not empty do
10:        Current = Q.pop()
11:        MaxRule = FindBest(Rules, StraightforwardRoot, Current.LockedTrue, Current.LockedFalse)
12:        if MaxRule == null then
13:            Continue
14:        NewNode = new Node(MaxRule);
15:        NewNode.Left = TerminalFalseNode
16:        NewNode.Right = TerminalTrueNode
       // Connect the parent node to current node
17:        if Current.ParentType == NoParent then
18:            Root = NewNode;
19:        else if Current.ParentType == TrueParent) then
20:            Current.ParentNode.Right = NewNode
21:        else if Current.ParentType == FalseParent then
22:            Current.Parent.Left = NewNode
23:        LeftChild = new QueueItem
24:        LeftChild.ParentNode = NewNode
25:        LeftChild.ParentType = FalseParent
26:        LeftChild.LockedTrue = Current.LockedTrue
27:        LeftChild.LockedFalse = Current.LockedFalse ∪ MaxRule
28:        Q.push(LeftChild)
29:        RightChild = new QueueItem
30:        RightChild.ParentNode = NewNode
31:        RightChild.ParentType = TrueParent
32:        RightChild.LockedTrue = Current.LockedTrue ∪ MaxRule
33:        RightChild.LockedFalse = Current.LockedFalse
34:        Q.push(RightChild)
35:    return Root
```

Algorithm 3 Finds the rule with the most significance to cost ratio in the boolean equation.

```
1: function FindBest(StraightforwardRoot, LockedTrue, LockedFalse)
2:     MaxRule = null
3:     MaxS = 0;
4:     for Rule in Rules do
5:         if Rule in LockedTrue or Rule in LockedFalse then
6:             Continue
7:         T = GetPassRatio(StraightforwardRoot, LockedTrue ∪ Rule,
   LockedFalse);
8:         F = GetPassRatio(StraightforwardRoot, LockedTrue, Locked-
   False ∪ Rule);
9:         S = |T - F| / Rule.Cost
10:        if Rule == null or MaxS < S then
11:            MaxS = S
12:            MaxRule = Rule
13:    return MaxRule
```

Algorithm 4 Returns the probability that the given decision tree returns true node

```
1: function GetPassRatio(Node, LockedTrue, LockedFalse)
2:     if Node.Type == TrueNode then
3:         return 1.0
4:     if Node.Type == FalseNode then
5:         return 0.0
6:     if Rule in LockedTrue then
7:         return GetPassRatio(Node.Right, LockedTrue, LockedFalse)
8:     if Rule in LockedFalse then
9:         return GetPassRatio(Node.Left, LockedTrue, LockedFalse)
10:    R = Node.Rule.Probability * GetPassRatio(Node.Right, LockedTrue
   ∪ Node.Rule, LockedFalse
11:    L = (1-Node.Rule.Probability) * GetPassRatio(Node.Left,
   LockedTrue, LockedFalse ∪ Node.Rule)
12:    Return R + L
```

Exact Brute Force Method

We also implemented a brute force method that searches through the entire solution space. This algorithm is expected to always give the global optimum in the case where cost and probability distributions are known. It will have a bad running time for filters with large amount of rules, but at least it will serve as a good comparison point for the quality of the other methods. The pseudo code of this implementation can be seen in algorithm 5.

The main difference to the boolean difference method is that instead of choosing the most significant solution, all the possible branches are considered. All rules that are not already picked and that can impact the boolean expression outcome are selected. The cost of selecting each of these possibilities as the next rule to be evaluated are then calculated recursively. The algorithm could be improved by pruning the search space and removing the infeasible solution candidates. By using the branch and bound algorithm we could derive the upper limit of the evaluation cost with faster methods and estimate the lower bound of any candidate solution from a incomplete solution through some method.

Algorithm 5 Creates the decision tree using complete traversal of the solution space. Requires access to a decision tree created using the straightforward or other method and the table of rules.

```
1: function CreateDecisionTreeExact(LockedTrue, LockedFalse)
2:     MinTree = new Tree;
3:     MinTree.Root = null
4:     MinTree.Cost = 0;
5:     for Rule in Rules do
6:         if Rule in LockedTrue or Rule in LockedFalse then
7:             Continue
8:         T = GetPassRatio(StraightForwardRoot, LockedTrue ∪ Rule, LockedFalse);
9:         F = GetPassRatio(StraightForwardRoot, LockedTrue, LockedFalse ∪ Rule);
10:        if T == F then
11:            Continue
12:        TBranch = CreateDecisionTree(LockedTrue ∪ Rule, LockedFalse)
13:        FBranch = CreateDecisionTree(LockedTrue, LockedFalse ∪ Rule)
14:        TotalCost = Rule.Cost + Rule.probability * TBranch.Cost + (1 - Rule.Probability) * FBranch.Cost
15:        if MinTree.Root == null or TotalCost < MinTree.Cost then
16:            MinTree.Root = new Node(Rule)
17:            if TBranch.root != null then
18:                MinTree.Root.Right = TBranch.root
19:            else
20:                MinTree.Root.Right = TerminalTrueNode
21:            if FBranch.root != null then
22:                MinTree.Root.Left = FBranch.root
23:            else
24:                MinTree.Root.Left = TerminalFalseNode
25:            MinTree.Cost = TotalCost
26:    return MinTree
```

Chapter 5

Results and Analysis

Simulations

Methods based on straightforward evaluation, rule set rearrangement, boolean difference calculus and the brute force method were all successfully implemented into the Tekla Structures source code to replace the old filter implementation. It is important to first test our methods in controlled simulations to see how they work in an environment similar to Tekla Structures models. The preferred method should give a significant improvement to the average running time. It should also not behave unexpectedly in special cases by for example having bad running for specific types of filters. One important aspect is to test how much the rule cost and the pass ratio statistics influence the performance for different methods. If for example, the pass ratio information doesn't give much value to the algorithm, we can then only use the cost information in the final implementation as the calculation of statistics always comes at a cost.

The simulations are performed using the new filter implementation that we will also use for testing with a real Tekla Structures model in the next section. In the simulations the filter is initialized with a list of pseudo rules, which follow a given probabilistic and cost distribution. The filter was then initialized and the average evaluation time was then calculated recursively from the decision tree according to algorithm 6.

Algorithm 6 Get the average evaluation cost of a decision tree

```
1: function GetCost(Node)
2:     if Node.type = TrueNode or Node.type = FalseNode then return
   0.0
3:     RightSide = Node.Rule.probability * GetCost(Node.RightChild)
4:     LeftSide = (1 - Node.Rule.probability) * GetCost(Node.LeftChild)
5:     return Node.cost + LeftSide + RightSide
```

Generation of Rule Sets

The cost distribution used in the simulations is inspired by the cost distribution of TS properties that that we investigated in section 2.3.4 and it is as follows:

- 50% of the rules will have cost 1
- 40% will have cost 10
- 10% will have cost 100.

The pass ratio for rules was chosen to be a random integer percentage value between $1 - 99\%$. The probabilities were statistically independent, which is not what one might find in a real customer model.

The rule count in generated rule sets varied from 1 to 30. Each rule was randomly assigned a cost and pass ratio picked from the above distributions. For each rule count a total of 1000 randomly picked rule sets were created. Each randomly picked rule set was then applied to filters of four different filter types.

Implemented filters rely on both cost and pass ratio information for optimal decision tree creation, with the exception of the straightforward method. The effect of the statistics information was also studied. For each filter type four separate filters were created. One filter used full knowledge of both cost and probabilistic information, one used only cost information, one used just the probability information and one used no statistics at all. If the filter has no cost information, it treats each property as having the same cost. If there is no probabilistic information, then the pass ratio of 0.5 is used for each property instead of the accurate ratio picked from the histogram.

Results

In figure 5.1 we can see the average evaluation cost for different filter methods as a function of the rule count. Both probabilistic and cost information was

used in the initialization of filters for this graph. It's clear that all the methods for improving the evaluation order work almost equally well in this case and they improve the performance significantly. This gives us a good sign that our methods were all correctly implemented. Brute force method is the best in all cases as it by definition should find the global optimum. The differences between methods however are minimal. Another interesting observation is that the average time it takes to evaluate an optimized filter seems to decrease as the the rule count increases for filters larger than six rules. This is due to the algorithm evaluating the least expensive and most significant values first to find a fast way to solve the filter.

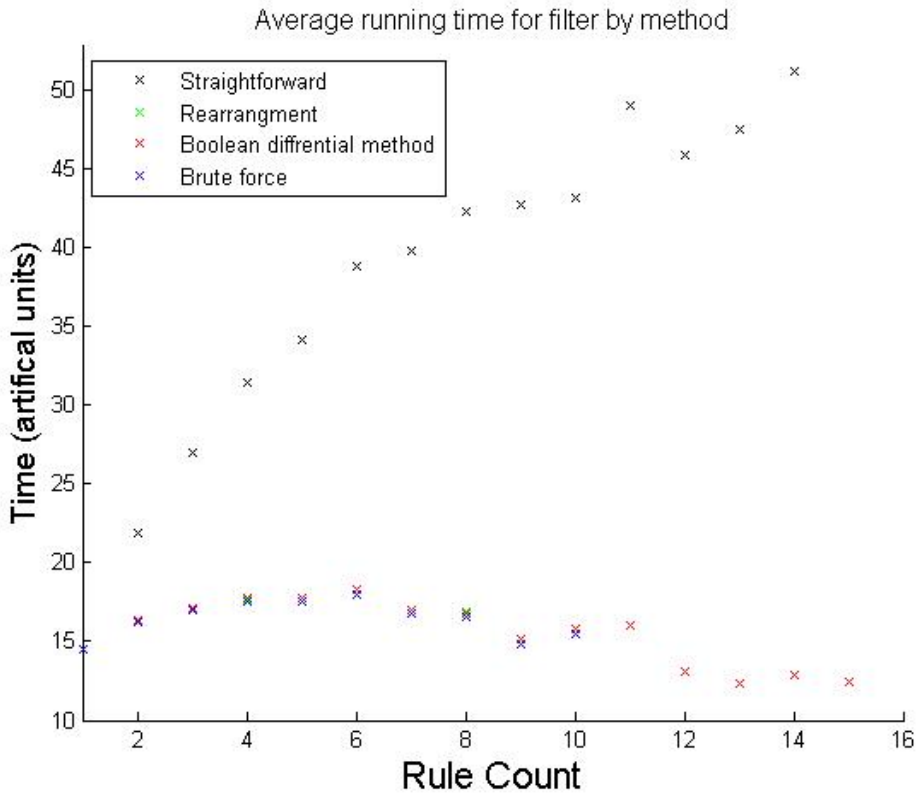


Figure 5.1: The average filter evaluation cost as a function of rule count. Full cost and probabilistic information where used for all the methods. The methods are so close that it is hard distinguish them in this plot.

To better compare the methods we can define a ratio of improvement

compared to the straightforward implementation as follows

$$Ratio_{Method} = \frac{T_{Straightforward} - T_{Method}}{T_{Straightforward}}. \tag{5.1}$$

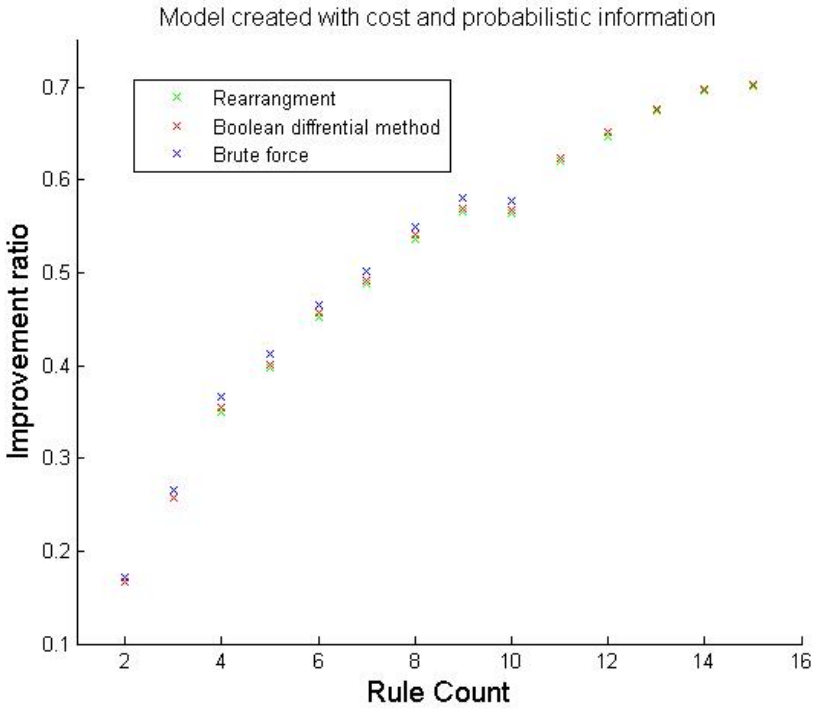
Using the average improvement ratio instead of the average time makes sure that filters with more costly rules will not dominate in the results. We also calculate the standard deviation of the improvement ratio as follows

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \quad \text{where} \quad \mu = \frac{1}{N}\sum_{i=1}^{N} x_i. \tag{5.2}$$

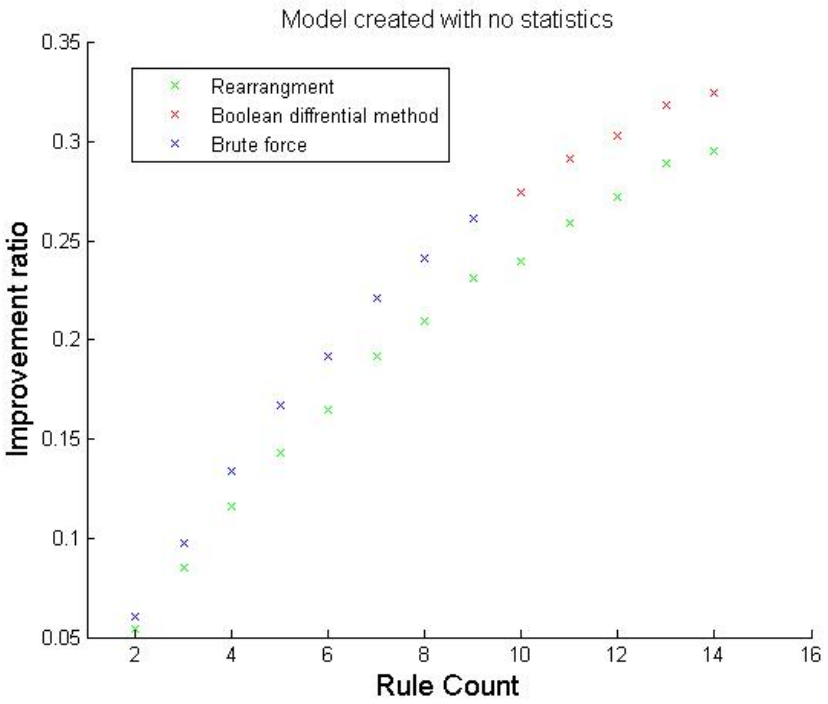
In figures 5.2 and 5.3 we can see the average improvement ratio of each filter type using different statistics information in decision tree creation. It's difficult to see differences in these graphs between different filter implementations, as the values are so close. The improvement ratios and their standard deviations can also can be seen in table form in appendix A where the small differences are more clear.

We can see that the performance is noticeably increased in all cases. The performance increases steadily as the number of rules increases, which is to be expected as larger filters have more to gain from optimized evaluation order. The brute force method works the best for all cases. The difference however is on average not more than 5% in any scenario compared to rearrangement and the boolean difference methods with the exception of the case where no statistics are used. The rearrangement method doesn't seem to work as well in the case where no statistics are used or where only probability information is used. The average improvement ratio is still only $5 - 15\%$ less than with the other methods.

One additional thing we can see is that in this case having the probability information doesn't seem to be as valuable as having the cost information. Having both the cost and the probability information gives only a few percent increase on top of the average total improvement ratio as compared to filter initialized with only cost information. This might be partially due to the chosen arbitrary probability distributions, which might not represent a real customer model. However, the correlation effects in the real models will also decrease the value of probabilistic information unless it is taken care of in the decision tree creation. The knowledge of the cost information gives a greater boost to the performance improvement ratio. It also works very well with all the tried methods.

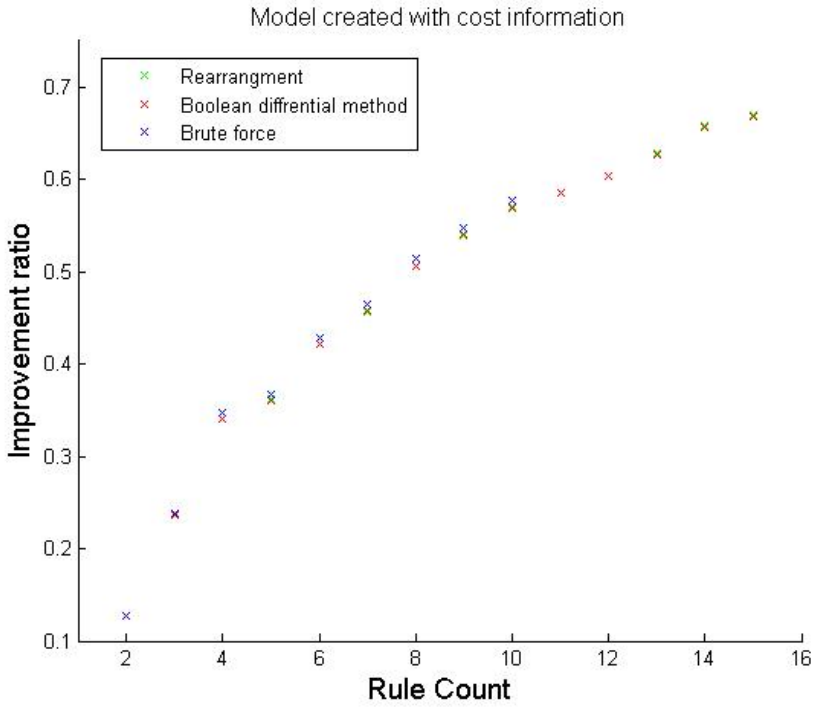


(a) Full cost and pass ratio information

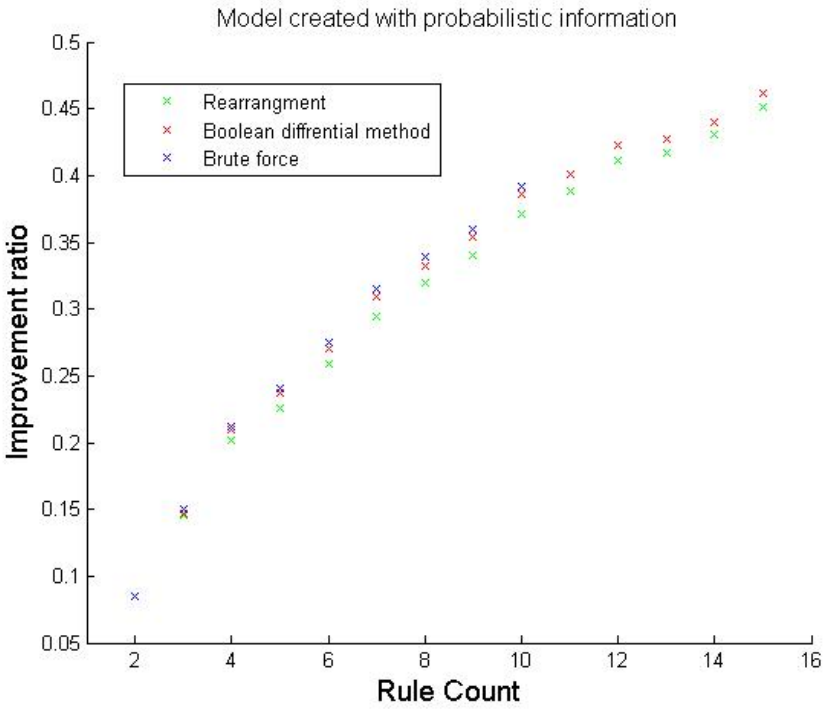


(b) No statistics infromation

Figure 5.2: Improvement of different methods compared to straightforward evaluation of the filter as a function of rule count.



(a) Full cost and no pass ratio information



(b) Full pass ratio and no cost information

Figure 5.3: Improvement of different methods compared to straightforward evaluation of the filter as a function of rule count.

Filter Creation Performance

The time to create and initialize the filter itself is also a very important metric in overall performance. Most of the time is spent in the creation of the decision tree from the set of rules according to the chosen heuristics. If the performance improvement given by optimized decision tree is only slight, using excessive time to create the tree is not optimal, unless it can be done in a background process or saved to disk for later use.

In table 5.1 we can see the average creation times of the filters. The values in the table are calculated from 1000 randomized rule sets using different methods and different amount of rules.

We can see that the straightforward and rearrangement methods are very fast, which is not surprising. The times for all the rule counts is about as fast as what the applying of the filter would be for one single model object. Boolean differential method is almost equally fast for all the sensible values of rule amount. It still uses only 1.7 ms for 30 rules, which is about the size of the largest filters found in our testing system. This means it should be a perfectly valid solution for filters of all sizes. Brute force method is computationally feasible for up to about ten rules. For ten rules the time is roughly two seconds, which is simply too much for typical scenarios. Filter needs to be applied perhaps millions of times to gain this time back if the the performance improvement is as small as we have measured compared to other models. However seven rules can still be processed in about $6ms$, which is still respectable but about 20 times more than what it takes for the other methods. This method thus would need a backup method for filters with 8 or more rules.

For comparison, the old filter implementation takes roughly $180\mu s$ for one rule filter, and $1.4ms$ for 10 rules. This however includes some validity checking that is not yet included in our model and the values are not directly comparable.

Table 5.1: The average creation time of the filter using different methods. All times are in μs

$\mathbf{N_{rules}}$	**Straightforward**(μs)	**Rearrangement**	**BD**	**Brute force**
1	3	3	3	3
2	4	5	6	6
3	5	6	8	13
4	6	8	11	43
5	14	14	21	182
6	14	17	25	1130
7	16	19	30	5710
8	16	19	38	37100
9	17	21	46	285000
10	19	23	56	1940000
15	43	42	167	-
20	53	53	411	-
25	63	66	886	-
30	100	123	1740	-

Tests With a Real Customer Model

The new filtering implementation was also tested with a real customer model visualized in figure 5.4. The model represented a large bridge consisting of more than two hundred thousand objects. 21646 objects were classified as parts, which include beams, plates, columns and slabs and generally any large single object in the model. The model also contains plenty of other object types such as bolts, welds or reinforcement bars. The testing focused on filtering by parts, as they are the most commonly used filter categories by our customers. They are also quite simple, as they don't require handling of the related objects. For example for the category assembly, the parent assemblies should also be fetched, which complicates the analysis.

Randomized Generation of Rule Sets

The generation of rule sets for testing with a real model is a somewhat difficult task. The amount of user created filters for models in the Farmi database is typically not large enough to gather meaningful statistics of performance improvement. Typically there is not much more than a dozen filters with more than one rule for each model and filters are typically not applicable to other models, as they may have completely different kinds of objects and

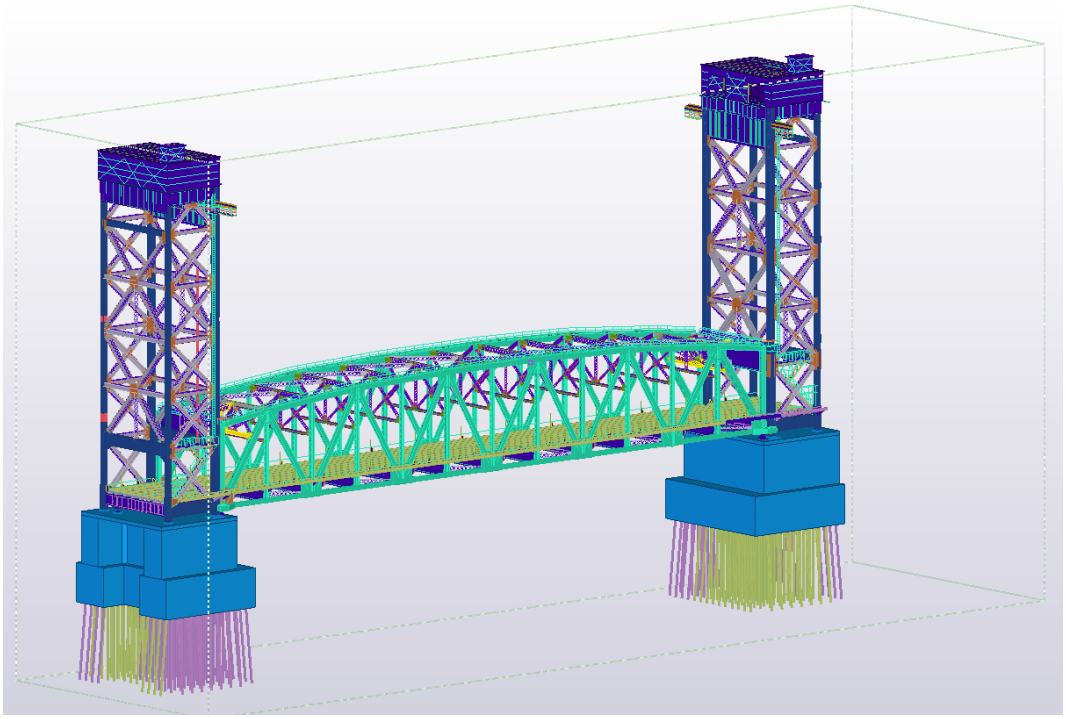


Figure 5.4: The model used in the tests was a real customer model representing a bridge. The model consisted of more than a hundred thousand objects.

property distributions. Of these few filters, some might already be ordered in a fast way, either by chance or by a very knowledgeable user, so that the improvement from optimal evaluation order is only slight, or there might even be no improvement at all.

A randomized way of creating filters was chosen as the most feasible for testing. The testing was done as follows. A set of 9 properties was chosen, which were evenly distributed to three groups: fast database properties($0.5-2\mu s$), middle properties($10-15\mu s$) and slow properties($> 40\mu s$). More properties should ideally be chosen for tests, but all the properties had to have values for a large percentage of the parts, which limited the selection. A histogram of values for these properties was then printed and five most common values were chosen for each property.

A Rule was chosen 50% time from fast properties, 10% of time from slow and 40% from middle properties. This will lead to the same property possibly appearing multiple times in the filter, but this is a characteristic of customer generated large filters. An expected value was then chosen by random from

the pre calculated set of five values. 80% of the time `and` conjunction was chosen between neighbouring filter rows. Only filters for which at least one model part passed the filter were approved for the test set. Number of rules in the filter varied from 1 to 10. Three different levels of statistics information were used. One used full cost and probability statistics, one used only cost statistics and one used only probability statistics. For every test case 500 approved filters were randomly created.

Results

Each of the generated filters was applied to all the model objects with the type `part`. The average time this took was calculated by a timer set in the source code of the test program. In figure 5.5 we can see the average filter evaluation time as a function of rule count for different methods. Both rule pass ratio and cost statistics were used in the initialization of filters in this graph.

We can see that the optimized evaluation order algorithms also improve the performance in the tests with a real Tekla Structures model. The execution time of the optimized filters remains roughly constant after three rules whereas the average time for the straightforward implementation grows steadily. It is once again difficult to distinguish the measure points in the graph, as all the algorithms give roughly similar results.

We also calculated the improvement ratio as in the equation 5.1 like we did for the earlier simulations. The results can be seen in table form in the appendix B and in figures 5.5 and 5.6.

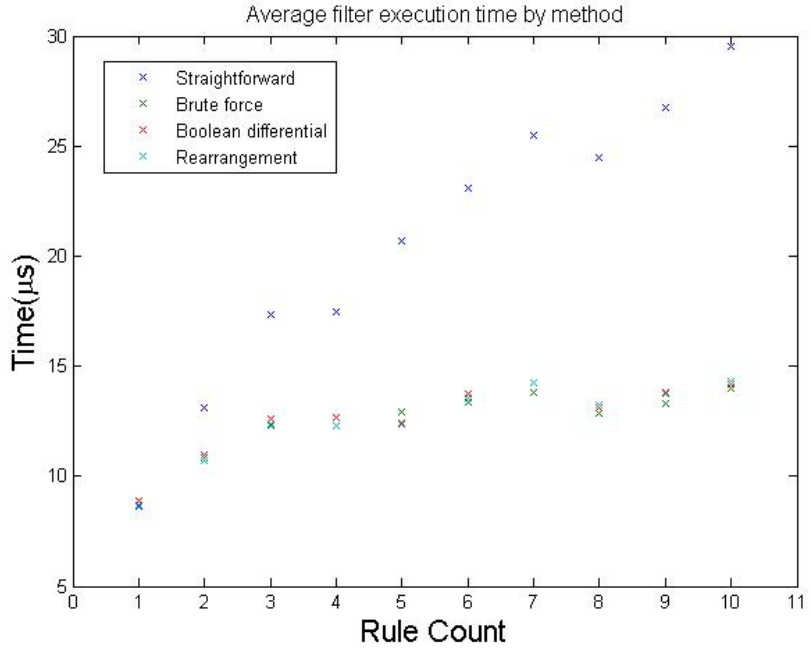
There is quite significant standard deviation in the improvement ratios that doesn't seem to depend on which algorithm is used. Using these results, it is not clear to distinguish that one method is clearly better than the other. As the standard deviation is sometimes larger than the improvement ratio this means that in some cases the effect of the optimization is actually negative. There is a number of things that might cause this and probably this is a combination of many things. One obvious point is the correlation effects, as the probabilities that we calculated are not statistically independent. The test environment may also not be very stable as the tests were run in a normal windows desktop computer. The time it took to run these tests was several hours and it is possible that some unrelated background processes caused some deviation in the results.

The brute force and the BD methods seem to lose about 3-4 percentage points compared to the rearrangement method already when the rule count is one or two. This has to be some kind of a systemic error in these calculations, as all the created decision trees should be practically equal when rule count

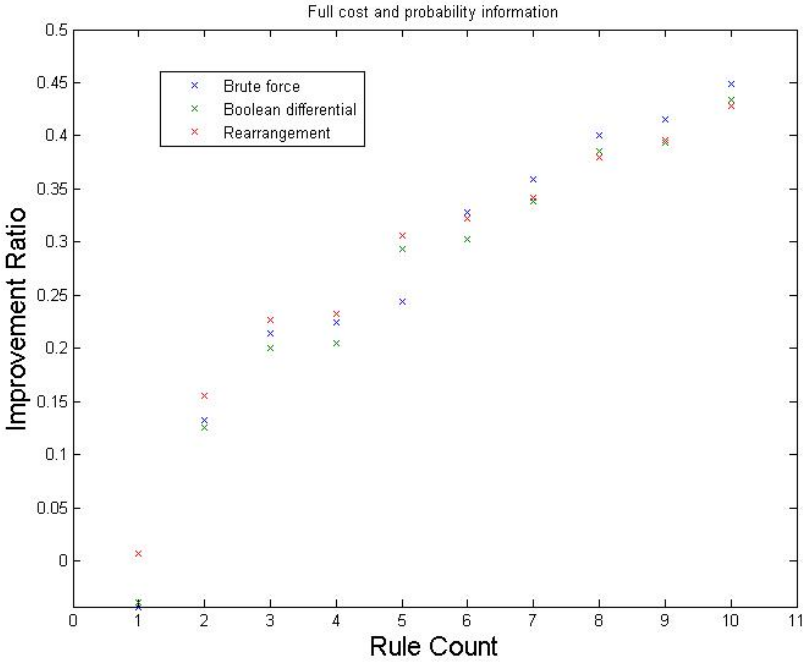
is only two and the same rule set was used for calculating both values.

In some of the rule sets the same property appeared many times. This can also cause some of the variance, as the number of properties that needed to be calculated was lower for some sets. For the BD and the brute force method an optimization method was implemented were the the cost of calculating a property was set to a very small value when the property was already calculated earlier in the evaluation for another rule. This might be the reason why they work better for large rule sets.

In figure 5.7 we can see the difference statistics information makes for the improvement ratio. The cost information seems to be once again more valuable than the pass ratio information. Having both cost and the probability statistics information seems to add approximately 4 percentage points difference to the improvement ratio compared to the case where only cost information is used.

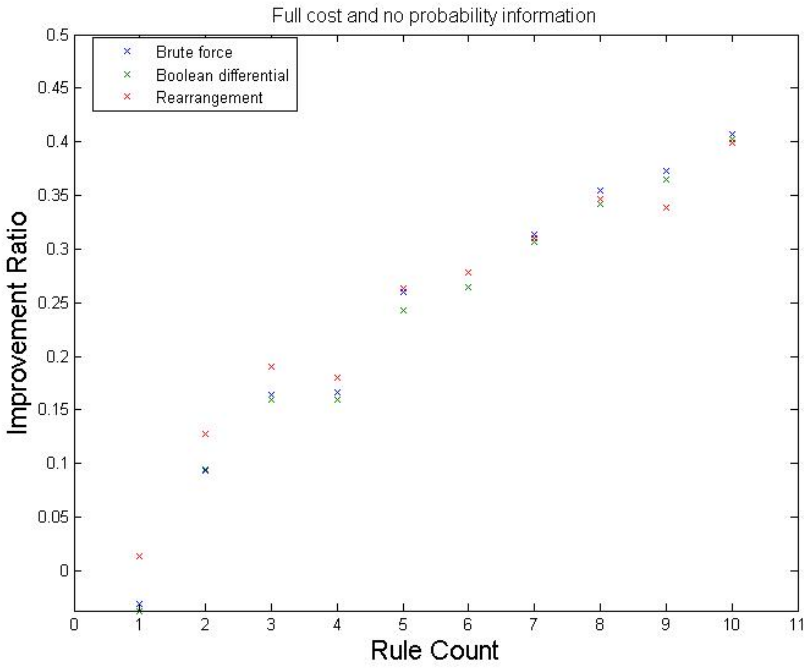


(a) The average evaluation time

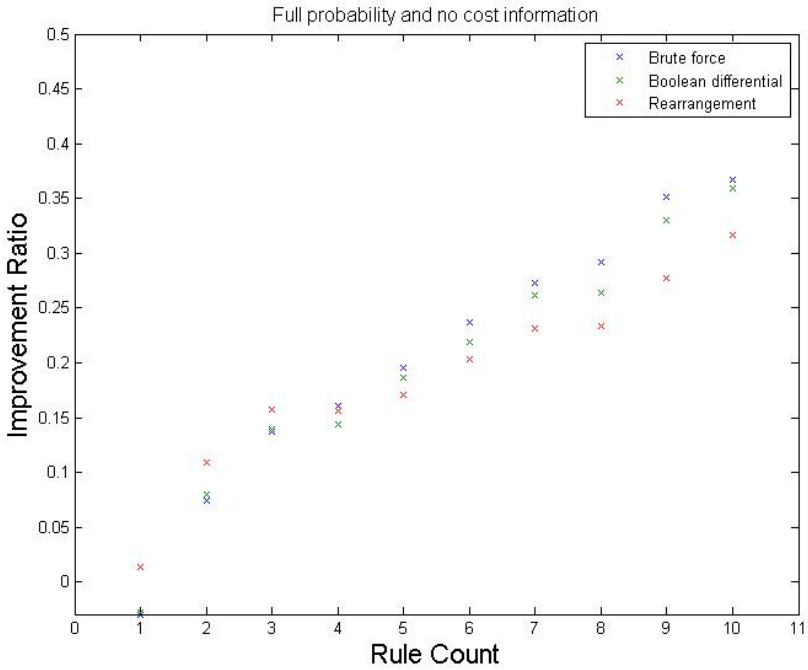


(b) The average improvement to straightforward evaluation

Figure 5.5: The average evaluation time of the filter as a function of rule count and the average improvement compared to the straightforward evaluation in the case where both cost and pass ratio statistics were used.



(a) Full cost information



(b) Full pass ratio information

Figure 5.6: Improvement ratio of different methods compared to straightforward evaluation of the filter as a function of rule count.

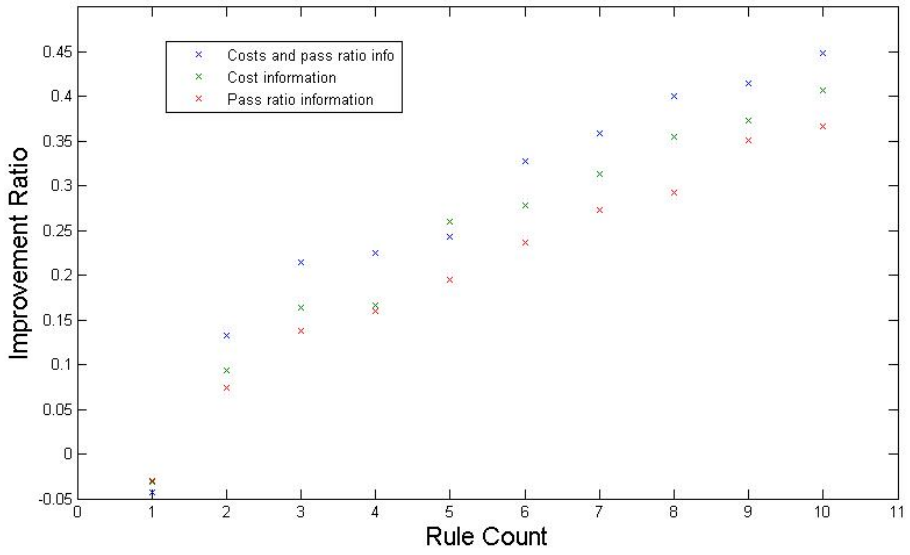


Figure 5.7: Comparison of how statistics information effects the improvement ratio as a function of the rule count. The brute force method was used for all filters.

Comparing Results to Original Implentation

The results were also calculated using the old filter implementation. The improvement ratio was calculated as

$$Ratio = \frac{T_{Original} - T_{New}}{T_{Original}}, \tag{5.3}$$

where $T_{Original}$ is the time it takes to run the original filter for all parts in the model and T_{New} is the same for for the new implementation. The average ratio was calculated from 500 randomly generated filters for each rule count from one to ten.

In figure 5.8 we can see the average difference to the original implementation. Two methods are visualized: the simple straightforward order and the brute force method. Even though the straightforward method works using the same principal as the old implementation and gives the same result to all the model objects the difference is still quite significant and the ratio of improvement only increases as the number of rules in the filter grows. The reason for this is the large overhead in the old equation library implementation. Let's look as an example a one rule filter checking the equality of part

`name` to certain expected value. It takes roughly $1.1\mu s$ for the new filter, which is only slightly more than what it takes to fetch the part `name`($0.5\mu s$) and object type($0.2\mu s$ from the database. For the old implementation the filter takes $6\mu s$ time, which which means there is a roughly $5\mu s$ overhead caused by the equation handling alone, which is almost completely eliminated with the new implementation. The performance improvement in this case is 82%, which makes a huge difference, given that name is the most used filter property. For the property `profile` the improvement is 'only' 15% as it takes $40\mu s$ to fetch from the property interface. Another big factor is storing already calculated values, if the same property is required for multiple rules. This was not done in the old implementation.

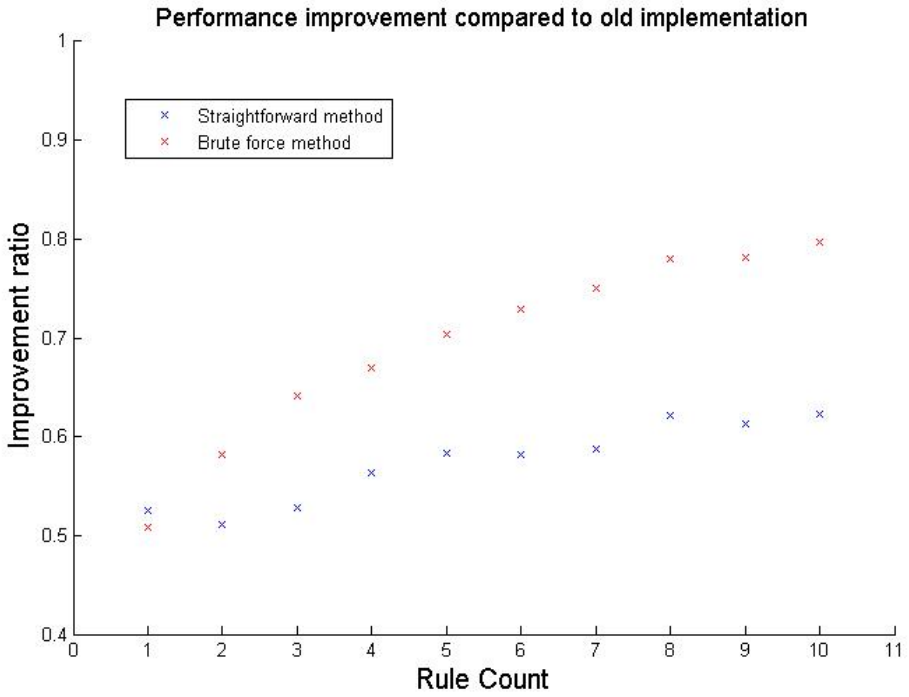


Figure 5.8: Performance improvement compared to the original implementation using. Both probabilistic and cost information are used for the brute force method.

Chapter 6

Conclusions

In this work we studied the current filtering implementation of Tekla Structures. We were able to find areas where the current implementation could be improved and studied new methods which could be used for this. This lead to the reimplementation of the equation processing in Tekla Structures and the optimization of the evaluation order of the filter rules was implemented as a new feature.

The performance improvement achieved by our work was quite significant and should be noticeable to our customers once it is merged to the customer version. The largest difference was made by the optimization of the equation processing by replacing the old equation library with a new implementation. This reduced the filtering cost by about 50% on average even when no evaluation order optimization was done. The size of this improvement was somewhat surprising, but serves as lesson that even relatively small changes can make a big difference in the optimization of real world applications. With the new equation processing implementation the performance of filtering is almost completely determined by the property calculation and not equation processing itself.

Most of the work focused on the improvement of the evaluation order of filters. It was shown that a significant performance improvement could be made if good statistics information is available. More work will be needed to provide this information during the program execution in a sensible way. The improvement is best for the large filters. It should also make the filtering speed more consistent, as the difference in evaluation time can be more than a order of magnitude in some cases if the filter is poorly constructed. In contrast, for some filters the performance might not improve or even slightly decrease if the available statistics are not sufficient or the user has been able to guess the right order. The average improvement is significant enough, that the evaluation order improvement should be implemented to the customer

version.

There is still work to be done to finalize the work we started. Extensive test cases need to be written for the new implementation. Additionally all the object categories and operators in the filter need to be supported in the filter evaluation. One special case that was not yet studied is the handling of the related objects for some categories, such as filtering by the `task` category. In this case all the related `task` objects assigned to a model object are compared.

Comparison of the Implemented Algoritms

All the algorithms for improving the evaluation order seemed to work very well for our software. In our tests with simulations and real customer models all algorithms were well within the standard deviation. The implemented algorithms were also very fast in the creation of the decision tree. The brute force method is the only one that requires a back up method for any feasible filter sizes and even it is able to handle vast majority of the filters. This leads us to believe that it is not worth it to implement new methods as the performance improvement will not be worth the effort. Randomized and genetic methods in particular generally require a number of iterations and random restarts to work well, and thus our methods with the exception of brute force search will likely outperform them in the filter initialization. A good method to determine the allowed moves between different solutions would be needed to provide improved results with randomized algorithms.

The best way to improve the performance of the algorithms could to be to improve on the performance of the brute force method by pruning infeasible solutions from the search space. This could be done by generating good lower and upper bounds for candidate solutions and using the branch and bound method instead of the full exhaustive search. The algorithm is already fast enough to deal with most of the filters in reasonable time, so improvement could bring it close to handling all but the few largest filters. One situation where the performance could be significantly improved is the case where the same property is used multiple times in different rules of the filter. As the value of the property is cached when the first of these rules is processed, the evaluation of the later rules becomes very fast. In our implementation this case was handled by setting the propery's cost to near zero for the later rules. Forcing these rules to be evaluated in sequential order would be very effective in pruning the search space for the exhaustive search.

The currently implemented algorithms are more than enough to provide a significant improvement and it will not be a worthy business goal to study more algorithms to improve the evaluation order. Another methods to opti-

mize the filtering performance should be considered such as optimizing the property interface.

Probability analysis

The overall impact of knowing the full distribution of probabilities of rules seemed relatively low to the end result. This is further discouraged by the fact that looping through the whole model to calculate accurate statistics is obviously not something that can happen every time we use the filter as was done for this test. The real life implementation would likely use a sampling method where only a randomly selected test set is used for probability estimations and this would be done perhaps daily or by user request. The filter is however often not operated on the whole model, but for some specific sub set of objects, so random sampling may not give good statistics information for the actual use case. Ideally perhaps the filter should learn these statistics on the fly from the model objects it is applied to by the user.

One area that could be the studied is the correlation effects between different rules. The cost-to-achievement ratio of this will probably not be enough though, as the correlations are expensive to estimate for filters with large number of rules.

Cost analysis

Significant improvement in the performance was achieved by using the cost statistics of properties. Implementing the cost based ordering seems to be a sensible business goal, but we will have to decide a way to get accurate statistics. The easiest way is to use hard coded values for property costs. The calculation time of properties doesn't seem to change much between different models, but a thorough study of this should be done and the calculation times should be examined. However, even just giving precedence to the few known fast database operations will almost certainly give a significant performance boost.

Optimization of the Property Management

In this work we studied the performance of property management, but our improvements did not focus on it. Property management is used very frequently throughout the source code and it is difficult to optimize in the scope of a project like this. Major changes will require large refactoring efforts and

extensive testing by a team of software engineers. However the property interface and common object hierarchy is currently undergoing refactoring and should be performing better in the future. This will also bring improvement in the filtering performance.

A large improvement in the process could be gained by doing most of the filtering in the database level and bypassing the property interface. In particular the user defined attributes, which make up approximately a third of the properties in filters are a great candidate for this. The database operation to fetch these properties takes only a fraction of microsecond, compared to the about $15\mu s$ overhead the property interface creates. Fetching this property from the database has the potential to improve the average filter running time very significantly. The work to implement this performance improvement has already started. In some cases also more than one property could be fetched from the database at once leading to more efficient database queries. Additionally the filtering could be implemented as working for a set of objects instead of processing objects just one by one. This would lead to more efficient database queries where a property is fetched for multiple objects at once.

As the number of cores in computer processors is ever increasing, parallel execution should be used for performance heavy operations. The filtering operation itself should be relatively easy to run in parallel. However the property interface currently can't be accessed in parallel execution and fixing this will require a lot of code refactoring efforts.[12].

Bibliography

[1] Tekla Structures website http://www.tekla.com/products/tekla-structures, Retrieved 10.5.2016

[2] Trimble company website http://www.trimble.com/, Retrieved 15.5.2016

[3] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on, 100(8), 677-691.

[4] Wegener, I. (1994). Efficient data structures for Boolean functions. Discrete Mathematics, 136(1), 347-372.

[5] Steinbrunn, M., Moerkotte, G., & Kemper, A. (1997). Heuristic and randomized optimization for the join ordering problem. The VLDB Journal - The International Journal on Very Large Data Bases, 6(3), 191-208.

[6] Vellev, S. (2009). Review of Algorithms for the Join Ordering Problems in Database Query Optimization. Information Technologies and Control, 1, 32-40.

[7] Schneeweiss, W. G. (2012). Boolean functions: with engineering applications and computer programs. Springer Science & Business Media.

[8] Kemper, A., Moerkotte, G., & Steinbrunn, M. (1992, August). Optimizing boolean expressions in object bases. In VLDB (Vol. 92, pp. 79-90).

[9] Documentation of the MariaDB usage of histogram statistics. https://mariadb.com/kb/en/mariadb/histogram-based-statistics/, Retrieved 15.5.2016

[10] Land, A. H., & Doig, A. G. (1960). An automatic method of solving discrete programming problems. Econometrica: Journal of the Econometric Society, 497-520.

[11] Clausen, J. (1999). Branch and bound algorithms-principles and examples. Department of Computer Science, University of Copenhagen, 1-30.

[12] Pirila, V., (2009). Improving Performance in a Large Legacy Software

Appendix A

Results of the Simulations

In this appendix we will show the results of the simulation runs.

Table A.1: The improvement ratio and their standard deviations of results when both probabilistic and cost statistics were used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
2	0.16696	0.05876	0.16671	0.05887	0.17133	0.05659
3	0.25757	0.07782	0.25717	0.07808	0.26507	0.07525
4	0.34925	0.09272	0.35439	0.09184	0.36562	0.08641
5	0.39822	0.08564	0.40116	0.08493	0.41199	0.08028
6	0.45256	0.08960	0.45661	0.08704	0.46572	0.08404
7	0.48786	0.08177	0.49109	0.08113	0.50124	0.07710
8	0.53656	0.07609	0.54109	0.07356	0.55025	0.07043
9	0.56687	0.07442	0.56956	0.07383	0.58125	0.06689
10	0.56415	0.07589	0.56758	0.07426	0.57702	0.07033
15	0.70197	0.05265	0.70338	0.05192	-	-
20	0.78288	0.03693	0.78362	0.03208	-	-
25	0.83062	0.02535	0.82979	0.02541	-	-
30	0.85574	0.02270	0.85514	0.02280	-	-

Table A.2: The improvement ratio and their standard deviations of results when only cost statistics was used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
2	0.12673	0.03967	0.12673	0.03967	0.12673	0.03967
3	0.23708	0.06237	0.23606	0.06319	0.23881	0.06250
4	0.34142	0.07218	0.34130	0.07283	0.34663	0.07112
5	0.36181	0.06945	0.36023	0.07055	0.36744	0.06849
6	0.42182	0.06682	0.42122	0.06824	0.42893	0.06568
7	0.45683	0.06357	0.45788	0.06386	0.46501	0.06172
8	0.50619	0.05909	0.50673	0.05991	0.51489	0.05726
9	0.56986	0.05523	0.54017	0.05568	0.54803	0.05332
10	0.56415	0.05567	0.56852	0.05655	0.57750	0.05351
15	0.66937	0.04364	0.66830	0.04400	-	-
20	0.74501	0.03273	0.74204	0.03399	-	-
25	0.80067	0.02152	0.79839	0.02219	-	-
30	0.83292	0.01681	0.83292	0.01739	-	-

Table A.3: The improvement ratio and their standard deviations of results when only pass ratio statistics was used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
2	0.08483	0.01294	0.08483	0.01294	0.08483	0.01294
3	0.14532	0.02441	0.14727	0.02457	0.14973	0.02381
4	0.20223	0.03081	0.20965	0.03043	0.21214	0.02962
5	0.22617	0.03249	0.23667	0.03083	0.24013	0.02999
6	0.25866	0.03260	0.27061	0.03180	0.27522	0.03074
7	0.29435	0.03632	0.30974	0.03418	0.31484	0.03290
8	0.31929	0.03847	0.33244	0.03898	0.33855	0.03689
9	0.34073	0.03963	0.35389	0.03712	0.35973	0.03530
10	0.37086	0.03829	0.38655	0.03785	0.39157	0.03634
15	0.45107	0.03718	0.46170	0.03596	-	-
20	0.50388	0.03674	0.50619	0.03735	-	-
25	0.54618	0.03567	0.54744	0.03600	-	-
30	0.56930	0.03432	0.56687	0.03732	-	-

Table A.4: The improvement ratio and their standard deviations of results when no statistics was used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
2	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
3	0.05411	0.00932	0.06077	0.01203	0.06077	0.01203
4	0.08481	0.01499	0.09784	0.01884	0.09784	0.01884
5	0.11644	0.01766	0.13385	0.02030	0.13385	0.02030
6	0.14303	0.01985	0.16692	0.02355	0.16692	0.02355
7	0.16448	0.02096	0.19173	0.02358	0.19173	0.02358
8	0.19154	0.02304	0.22097	0.02440	0.22097	0.02440
9	0.20963	0.02424	0.24136	0.02492	0.24136	0.02492
10	0.23162	0.02529	0.26151	0.02500	0.26151	0.02500
15	0.29562	0.02719	0.32489	0.02599	-	-
20	0.32903	0.03012	0.34880	0.03053	-	-
25	0.37236	0.02962	0.38763	0.03009	-	-
30	0.38019	0.02999	0.39229	0.03059	-	-

Appendix B

Tests With a Customer Model

Table B.1: The improvement ratio and their standard deviations of results when both cost and pass ratio statistics were used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
1	0.0073	0.0795	-0.0387	0.0700	-0.0432	0.0305
2	0.1553	0.2902	0.1252	0.2984	0.1328	0.2776
3	0.2266	0.3157	0.2002	0.3206	0.2141	0.3113
4	0.2322	0.3173	0.2044	0.3516	0.2246	0.3070
5	0.3062	0.3420	0.2940	0.3452	0.2438	0.4809
6	0.3222	0.3254	0.3024	0.3441	0.3281	0.3189
7	0.3413	0.3263	0.3383	0.3278	0.3586	0.3121
8	0.3799	0.3390	0.3853	0.3316	0.4002	0.3187
9	0.3962	0.3339	0.3936	0.3457	0.4154	0.3244
10	0.4276	0.3345	0.4343	0.3292	0.4485	0.3205

Table B.2: The improvement ratio and their standard deviations of results only cost statistics was used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
1	0.0135	0.0293	-0.0370	0.1537	-0.0306	0.0273
2	0.1275	0.2968	0.0951	0.3081	0.0940	0.3076
3	0.1908	0.3289	0.1591	0.3416	0.1640	0.3398
4	0.1802	0.3550	0.1600	0.3540	0.1669	0.3521
5	0.2633	0.3614	0.2426	0.3766	0.2603	0.3599
6	0.2781	0.3559	0.2648	0.3647	0.2786	0.3585
7	0.3101	0.3495	0.3073	0.3481	0.3134	0.3513
8	0.3465	0.3666	0.3424	0.3589	0.3549	0.3490
9	0.3391	0.6829	0.3644	0.3639	0.3732	0.3526
10	0.3996	0.3624	0.4026	0.3575	0.4075	0.3532

Table B.3: The improvement ratio and their standard deviations of results when only pass ratio statistics was used were used.

Rule count	Rearrangement	σ	BD	σ	Brute Force	σ
1	0.0133	0.0571	-0.0280	0.0256	-0.0295	0.0272
2	0.1096	0.4233	0.0795	0.4390	0.0746	0.4518
3	0.1570	0.4092	0.1391	0.4110	0.1374	0.4091
4	0.1559	0.4836	0.1439	0.4907	0.1606	0.3972
5	0.1710	0.5990	0.1861	0.5648	0.1955	0.5439
6	0.2034	0.5395	0.2191	0.4834	0.2372	0.4756
7	0.2314	0.5383	0.2615	0.5208	0.2731	0.4687
8	0.2338	0.6162	0.2635	0.5974	0.2920	0.5397
9	0.2773	0.4399	0.3298	0.3981	0.3516	0.3463
10	0.3170	0.4759	0.3588	0.4215	0.3668	0.4190